

Lab 4: Generics: Stack, Reverse Polish evaluation and Unit testing

Objectives

- Getting familiar with Generics in Java programming language
- Applying them in a Java project to write generic classes, methods, ...
- Getting familiar with unit testing (JUnit)
- Full mark: 40 points

Source files

- `GenericStack.java`
- `GenericTrie`
 - `GenericTrie.java`
 - `GenericTrieDoubleTest.java`
 - `GenericTrieIntegerTest.java`

1 Introduction

Generics, or parameterized types, are a facility of generic programming that was added to the Java programming language in 2004 as part of Java 5 (JDK 1.5). They allow "a type or method to operate on objects of various types while providing compile-time type safety". A common usage of this feature is when using a Java *Collection* that can hold objects of any type, to specify the specific type of object stored in it. In this lab, we try to address Generics for containers, such as vectors and stacks.

A Java collection (we will explore more in Lab 5) is a flexible data structure that can hold heterogeneous objects where the elements may have any reference type. It is your responsibility, however, to keep track of what types of objects your collections contain. For example, consider adding an integer to a collection: since you cannot have collections of primitive data types you must convert it to the corresponding reference type (i.e. `Integer`) before storing it in the collection.

1.1 The Motivation

The motivation for adding generics to the Java programming language stems from the lack of information about a collection's element type, the need for developers to keep track of what type of elements collections contain, and the need for casts all over the place. Using generics, a collection is no longer treated as a list of object references, but you would be able to differentiate between a collection of references to integers and collection of references to bytes. A collection with a generic type has a type parameter that specifies the element type to be stored in the collection. So, instead of relying on the programmer to keep track of object types and performing casts, which could lead to failures at **runtime**, the compiler can now help the programmer enforce a greater number of type checks and detect more failures at compile time.

This is a quick example of this application of Generics in the casting problem.

```
List<Apple> box;  
// Put some apples in the box  
Apple apple = box.get(0);
```

In the previous code, `box` is a reference to a list of objects of type `Apple`. The `get` method returns an `Apple` and no casting is required. Without generics, this code would have been:

```
List box;  
// Put some apples in the box  
Apple apple = (Apple) box.get(0);
```

The main advantage of generics is having the compiler keep track of type parameters, perform the type checks and the casting operations. The compiler guarantees that the casts will never fail.

1.2 The Generic Facility

The generics facility introduced the concept of type variable. A type variable, according to the Java language specification, is an unqualified identifier introduced by:

- Generic class declarations
- Generic interface declarations
- Generic method declarations
- Generic constructor declarations

Moreover, other Java concepts can be used in a generic manner:

- A **class** is generic if it declares *one or more* type variables. These type variables are known as the type parameters of the class. All of these parameterized types share the same class at runtime:

```
public class Foo<T> { }
```

- An **interface** is generic if it declares **one or more** type variables. These type variables are known as the type parameters of the interface. Similar to the above, all parameterized types share the same interface at runtime:

```
public interface Foo<T> { }
```

- A **method** is generic if it declares **one or more** type variables. These type variables are known as the formal type parameters of the method. The formal type parameter list is identical to a type parameter list of a class or interface:

```
public <T> T getT() { }
```

- A **constructor** can be declared as generic, independently of whether the class itself is generic or not. A constructor is generic if it declares **one or more** type variables. These type variables are

known as the formal type parameters of the constructor. The formal type parameter list is identical to a type parameter list of a generic class or interface:

```
public class Foo<T> {  
    public <E> Foo(E e) { }  
}  
new Foo<Integer>("I'm a string");
```

1.3 Bounded Types

There may be times when you want to restrict the types of the generics. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what bounded type parameters are for. To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, and then followed by its *upper bound*, which, in this example, is `Number`. Note that in this context, extending is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

```
<T extends SomeSuperClassOrInterface>
```

2 Stack

The stack represents a *last-in-first-out* (LIFO) list of objects. It extends `Vector` with five operations that allow a `Vector` to be treated as a stack:

- `peek`: query the top element of the stack. **The top element is the only accessible element.** If the stack is empty, then you can peek nothing.
- `push`: add a new element to the top of the stack. If the stack has a max size and is full, then you cannot successfully push.
- `pop`: remove the top element of the stack. If the stack is already empty, then you cannot successfully pop.
- `size`: query the size of the stack.
- `isEmpty`: query if the stack is empty or not.

3 Postfix Expression

Postfix notation, also known as *reverse Polish notation*, is a syntax for mathematical expressions where the mathematical operator is always placed after the operands. For instance, the addition of 1 and 2 would be written in postfix notation as `1 2 +`. Computer scientists and software developers are interested in postfix notation because it can be easily and efficiently evaluated with a simple stack machine. Moreover, postfix notation has been used in some hand-held calculators because it enables arbitrarily complex expressions to be entered and evaluated without the use of parentheses and without requiring the user to keep track of intermediate results.

Here are some simple postfix expressions and their results.

Postfix Expression

Result

4 5 +	9
9 3 /	3
17 8 -	9
6 2 / 5 +	8

4 Unit testing

JUnit is an open source Unit Testing Framework for JAVA. It is useful for Java Developers to write and run repeatable tests. Once you are done with code, you should execute all tests, and it should pass. Every time any code is added, you need to re-execute all test cases.

Unit Testing and its importance can be understood by below-mentioned points:

- Unit Testing is used to identify defects early in software development cycle.
- Unit Testing will compel to read our own code. i.e. a developer starts spending more time in reading than writing.
- Defects in the design of code affect the development system. A successful code breeds the confidence of developer.

The following links are for getting familiar with Junit.

<https://www.guru99.com/junit-tutorial.html>

https://www.tutorialspoint.com/junit/junit_test_framework.htm

5 Deliverable 1 -- Postfix Expression Calculator by Generic Stack

Step 1:

Refer to the *GenericStack.java*, and finish the five methods of stack operations.

Step 2:

Finish the `calcPostfixExpression` method following the brief algorithm:

- Initialize the stack.
- Parse the expression string into an array of symbols, and feed them to the stack one by one.
- If the next symbol is an operand (numbers), then push it to the stack.
- If the next symbol is an operator, then there must be at least two operands already in the stack. Pop the two operands to do the corresponding calculation, and then re-push the result back to the stack.
- Repeat through all the symbols.
- Finally, we have only one element in the stack -- it is the final result. Pop and print it out.

6 Deliverable 2 -- Generic Trie

Make your Trie data structure generic and associative. Please refer to the *GenericTrie.java*, and finish all the TODO codes. Note: you also need to implement the remove method. After finishing your code, you need to test the code using JUnit. Test your code using *GenericTrieIntegerTest.java* and *GenericTrieDoubleTest.java* which are in the “test” folder. The test cases in *GenericTrieDoubleTest.java* need to be implemented following the same ideas as *GenericTrieIntegerTest.java*.

Lab 4: Generics: Stack, Reverse Polish evaluation and Unit testing

Marking sheet

Deliverable 1 -- Postfix Expression Calculator by Generic Stack

GenericStack implementation

	Attributes defined but with fix size storage (1)	Class correctly defined (attributes and constructor) to support dynamic storage (2)		/2
--	--	---	--	----

Implementation <code>peek</code> works correctly	/1
--	----

Implementation <code>push</code> works correctly	/1
--	----

Implementation <code>pop</code> works correctly	/1
---	----

Implementation <code>size</code> works correctly	/1
--	----

Implementation <code>isEmpty</code> works correctly	/1
---	----

calcPostfixExpression implementation

	Stack is correctly created and filled, but the evaluation is wrong (1-2)	Stack is correctly implemented and the calculation is correct (3-5)		/5
--	--	---	--	----

Deliverable 2 -- Generic Trie

Implementation <code>search</code> works correctly	/3
--	----

Implementation <code>insert</code> works correctly	/4
--	----

Implementation of <code>remove</code>	/7
---------------------------------------	----

Implementation <code>startsWith</code> works correctly	/3
--	----

Implementation of <code>test cases for Double</code> works correctly	/5
--	----

Code quality

Naming and usage of variables (in both deliverables)	/2
--	----

Documentation (in both deliverables)

	No documentation (0)	One line comments but not using proper javadoc (1-2)	Clear documentation about what the function /procedure does using javadoc style (3-4)	/4
--	----------------------	--	---	----

			Total:	/40
--	--	--	---------------	------------